Infraestructura para la comunicación entre componentes Java en el estilo arquitectónico C2*

Enrique Soriano Salvador[1]**
Isabel Muñoz Fernández[2]
Jorge Enrique Pérez Martínez[2]

[1] Departamento de Informática, Estadística y Telemática, ESCET, Universidad Rey Juan Carlos,
C. Tulipán S/N, 28933 Móstoles (Madrid), España.
esoriano@gsyc.escet.urjc.es
[2] Departamento de Informática Aplicada, EUI, Universidad Politécnica de Madrid,
Ctra. de Valencia Km. 7, 28031 Madrid, España.
{imunoz,jeperez}@eui.upm.es

Resumen La arquitectura del software es una disciplina dentro de la ingeniería del software que ofrece una descripción de alto nivel de la estructura de las aplicaciones, normalmente definida en términos de componentes, conectores y restricciones topológicas. En este contexto, el estilo arquitectónico C2 ha sido propuesto para arquitecturas altamente distribuidas.
En este trabajo se describe el diseño y la implementación de una infraestructura para la comunicación entre componentes que sigan el estilo arquitectónico C2 sobre una plataforma Java. Un requisito de esta infraestructura es que componentes y conectores se ejecuten cada uno en su propia máquina virtual (JVM) en el mismo nodo o en nodos diferentes. Se ha diseñado un conjunto de clases que proporcionan mecanismos para la comunicación entre componentes y conectores C2. Como parte del trabajo, se han evaluado las tecnologías disponibles para Java que permiten construir la infraestructura, habiéndose elegido la invocación remota a método (RMI) como la base para la comunicación entre los componentes del sistema.

1. Introducción

El estándar 1471 de IEEE [5] define la arquitectura software como la organización fundamental de un sistema encarnada en sus componentes, las relaciones entre ellos y el entorno, y los principios que guían su diseño y evolución. La arquitectura de una aplicación puede seguir alguno de los múltiples estilos arquitectónicos que han sido descritos en la literatura [1,12].

* Parcialmente financiado por el MCyT (TIC2001-1586-C03-01). Deseamos agradecer a Antonio Fernández y a Sergio Arévalo Viñuales su supervisión y aportaciones.
** El trabajo de este autor ha sido financiado por el MCyT con la beca FPI BES-2003-2942.

En este trabajo se describe el diseño y la implementación de una infraestructura para la comunicación de componentes en Java que siga el estilo arquitectónico C2 [6,13]. En esta infraestructura, un componente tendrá sus propios flujos de control y deberá ser un proceso pesado. Dado que la plataforma elegida es Java, esto implica que cada componente deberá ejecutarse en su propia máquina virtual. Se ha elegido el modelo de procesos pesados frente al modelo de procesos ligeros por dos razones: (1) la disminución de rendimiento de una máquina virtual de Java cuando tiene que soportar un número elevado de threads [2]; (2) la fiabilidad en los componentes, dado que el fallo de una máquina virtual no supone el fallo de todo el sistema, sino el fallo de un solo componente o conector.

Para la construcción de la infraestructura se han evaluado distintas tecnologías disponibles para Java que permiten la creación de componentes y la comunicación entre procesos Java. En el primer grupo se encuentran Java Beans [8] y Enterprise Java Beans (EJB) [11]. En el segundo grupo se encuentran Java Messaging Service (JMS) [9] y Remote Method Invocation (RMI) [10].

En este trabajo se detalla además la implementación de la solución elegida. Esta permite la creación de la infraestructura de componentes basada únicamente en la tecnología Java RMI y en los mecanismos de introspección de Java Beans.

El resto del documento se estructura como sigue. En la sección 2 se describe brevemente el estilo arquitectónico C2. La sección 3 contiene un breve análisis de las tecnologías Java existentes para el desarrollo basado en componentes. En la sección 4 se analizan las capacidades de Java RMI y JMS para soportar la comunicación intercomponente en el estilo arquitectónico C2. Este análisis se ilustra mediante el estudio de un problema utilizado comúnmente para la evaluación de arquitecturas software: el problema Meeting Scheduler. La sección 5 presenta la infraestructura que hemos desarrollado para soportar la comunicación entre componentes C2 utilizando Java RMI. Por último en la sección 6 se describen las principales conclusiones de este trabajo y futuras líneas de investigación.

2. El estilo arquitectónico C2

C2 es un estilo arquitectónico que se puede definir como una red de componentes concurrentes unidos por dispositivos de encaminamiento de mensajes [6]. En el estilo arquitectónico C2 un componente se conecta a otros componentes mediante conectores. Un conector se puede conectar a un componente por su parte inferior (dominio *bottom*) o por su parte superior (dominio *top*). Mediante un conector, un componente estará conectado por su parte inferior a la parte superior de otro componente y/o estará conectado por su parte superior a la parte inferior de otro componente. Un mismo conector puede conectar dos o más componentes.

Un componente envía las peticiones a otros componentes mediante dos tipos de mensajes: *requests* y *notifications*. Las *requests* son peticiones que se envían por el dominio *top* de un componente y se reciben por el dominio *bottom* de otro

(mediante uno o más conectores). Las *notifications* son peticiones que se envían por el dominio *bottom* de un componente y se reciben por el dominio *top* de otro componente. En el estilo C2 las *requests* atraviesan la arquitectura hacia arriba mientras que las *notifications* lo hacen hacia abajo.

Los componentes procesan las peticiones de forma asíncrona. Por tanto el emisor de una petición no queda bloqueado hasta que esta termine de procesarse. Además, las peticiones no generan valores de retorno: si un objeto necesita comunicar algún dato después de procesar una *request*, deberá emitir una *notification*.

Los componentes no se conocen entre ellos. Un componente pide a un conector un servicio pero no se preocupa de qué componente lo oferta. La tarea de un conector es precisamente encaminar esas peticiones de servicio y eliminar las referencias entre los componentes. Los conectores distribuirán las peticiones de servicio siguiendo una política de filtrado. Existen varias políticas de filtrado, pero las más utilizadas son las políticas *no filtering* y *message filtering*. La primera obliga a que todos los mensajes lleguen a los componentes, aunque estos no los entiendan. La segunda obliga a que ningún mensaje llegue a un componente que no lo entienda.

Los autores del estilo arquitectónico C2 ofrecen un *framework* para Java, llamado ArchStudio 3.0 [4], que permite la especificación de arquitecturas software basadas en componentes y conectores, expresados en diferentes estilos arquitectónicos entre los que se encuentra C2. Hemos estudiado la implementación de ArchStudio [3] y comprobado que es un *framework* muy flexible, pero a la vez demasiado complejo. Por ello, hemos optado por la implementación de una infraestructura a medida y con un diseño más sencillo para la comunicación de componentes. Durante el desarrollo de nuestra infraestructura, ArchStudio ha sido considerado como una implementación de referencia de C2.

3. Componentes en Java

En este trabajo se han estudiado distintas tecnologías de Java para la implementación de una infraestructura para comunicar componentes. Estas tecnologías se pueden agrupar en dos conjuntos: tecnologías de componentes en Java y tecnologías de comunicación entre procesos. Varias de estas tecnologías son compatibles entre sí y se pueden usar de forma conjunta para crear los componentes. Incluso algunas de ellas utilizan a otras en su implementación. A pesar de ello, se han evaluado aisladamente una de otra. Por ejemplo, cuando se hable de Java RMI, se entenderá que los componentes tan sólo usarán RMI como medio de comunicación y que la infraestructura para su creación la implementaremos nosotros mismos.

Lo primero que hemos observado es que el concepto de componente es muy amplio y el componente C2 es distinto de los componentes que ofrecen las tecnologías existentes para Java. Como se indicó en la introducción, las tecnologías para la creación de componentes consideradas han sido Java Beans y Enterprise Java Beans.

Los Java Beans son componentes persistentes que están ideados para ejecutarse en una única máquina virtual de Java y para comunicarse por eventos dentro de ella (o por invocación local, como cualquier clase de Java). En un principio, no precisamos que nuestros componentes sean persistentes, por tanto esa cualidad de los Java Beans tampoco es relevante para nuestra elección. El aspecto más interesante de los Java Beans es la posibilidad de introspección de las clases. Sin embargo, esos servicios no necesitan usar Java Beans como componentes, sino que pueden ser utilizados por cualquier clase Java que importe la API de Java Beans e instancie un ejemplar de la clase encargada de hacerlo. Por ejemplo, el prototipo propuesto en la sección 5 de este artículo usa las propiedades de introspección de Java Beans.

Por su parte, EJB (Enterprise Java Beans) es un *framework* de J2EE para el desarrollo de aplicaciones distribuidas. EJB proporciona una especificación estándar para incorporar la lógica de la aplicación y la lógica del negocio a un sistema que ofrezca la lógica del sistema, facilitando la reutilización del código. Este sistema se denomina *contenedor* de EJB. EJB está orientado a la creación de componentes que formen aplicaciones multicapa (*N-Tier*) en las que cada capa se ocupa de una lógica diferente (sistema, negocio o aplicación). En particular, el contenedor de EJBs ofrece la lógica del sistema.

Sin embargo, EJB no se adecuan a nuestras necesidades. Uno de los objetivos finales de nuestra infraestructura es dar soporte para la implementación de un sistema que ofrezca los bloques básicos de construcción en sistemas tolerantes a fallos. Por tanto, el sistema de componentes en ese caso deberá ofrecer la lógica del sistema (servicios de detección de fallos, de consenso, de canales fiables, de canales no fiables, etc.) a otros componentes que implementarán la lógica del negocio y la de la aplicación. Por esa razón creemos que EJB ofrece una infraestructura para construir componentes de más alto nivel de abstracción que los que se podrían querer desarrollar con nuestra infraestructura. Por ejemplo, los componentes del sistema para tolerancia a fallos no necesitan tener acceso a bases de datos ni usarán transacciones, ya que son abstracciones del mismo nivel: en el propio sistema habrá un componente que ofrezca el servicio de transacciones como un bloque básico de construcción para sistemas tolerantes a fallos.

4. Panorámica de las soluciones

Además de las tecnologías que ofrecen una infraestructura para la creación de componentes, existen otras que permiten comunicar procesos. Si implementamos cada componente C2 como un proceso pesado Java, podemos utilizar estas tecnologías para la comunicación entre componentes. En esta sección se describen dos posibles soluciones para comunicar componentes en el estilo C2 utilizando dos tecnologías Java: RMI y JMS.

4.1. Esquema básico con Java RMI

Java RMI proporciona invocación remota a métodos y puede utilizarse para la comunicación intercomponente, de tal forma que las peticiones entre compo-

nentes se procesen como invocaciones remotas. La invocación remota por RMI es síncrona. Por tanto, el método que invoca una operación remota se queda bloqueado hasta que el método invocado acaba su ejecución y devuelve el valor de retorno (si es que existe). Este comportamiento entra en conflicto con una de las propiedades del estilo arquitectónico C2, en el cual ningún componente se queda bloqueado esperando la devolución del control. Para solucionar este problema, se puede implementar un mecanismo de invocación por RMI que se comporte de manera asíncrona. La implementación de invocaciones remotas asíncronas con RMI es relativamente sencilla. Los componentes procesarán las peticiones con métodos que se encarguen de almacenarlas en colas, y que devuelvan el control inmediatamente. Otro *thread* de ejecución en el componente se encargará de procesar realmente las invocaciones, procesando las peticiones encoladas.

Con RMI, un objeto debe tener una referencia a los otros objetos a los que pide servicio. Para evitar que un componente tenga referencias a otros componentes que le ofrecen servicio, en nuestra solución hemos introducido otros procesos conectores que se encarguen de encaminar las peticiones. Estos procesos conectores también se encargarán de aplicar la política de filtrado de C2. Los componentes tendrán una referencia a sus conectores superior e inferior, y el encargado de enlazar conectores con componentes será un proceso monitor.

Ejemplo del Meeting Scheduler con Java RMI Para evaluar la solución descrita se ha implementado una versión simplificada de un Meeting Scheduler [14,7], considerado un problema modelo para la evaluación de arquitecturas software. El problema supone varios asistentes a una reunión (Attendees) que pueden ser de dos tipos: normales o importantes. Mediante el uso de un componente llamado Meeting Initiator, estos asistentes se deben poner de acuerdo en una fecha, un lugar y una lista de materiales necesarios para celebrar una reunión. Cada asistente tiene una lista de fechas preferentes y otra lista de fechas ocupadas para la reunión. Además, los asistentes importantes tiene una lista de lugares preferentes para la reunión. El componente Meeting Initiator deberá elegir los datos para la reunión y comunicárselos a todos los asistentes. Si existen conflictos entre las fechas se deberán solventar, por ejemplo, retirando algún asistente o ampliando el rango de fechas. La arquitectura del ejemplo se describe en la figura 1.

Como ya se ha comentado en el punto anterior, los componentes C2 sólo se comunican a través de conectores. Los conectores tienen referencias (ya sean referencias de Java o las URL de los elementos con los que se tienen que comunicar) a los conectores y componentes a los que están conectado. Un monitor se encargará de gestionar las conexiones entre los elementos del sistema, de tal forma que en cualquier momento pueda pararse la ejecución de los componentes y se pueda reconfigurar dinámicamente el sistema. El monitor es el único proceso que se comunica con todos los elementos del sistema. Cada elemento del sistema se deberá registrar en el registro de RMI (*rmiregistry*) de su máquina local. En la figura 2 se ilustra la arquitectura del ejemplo del Meeting utilizando Java RMI.

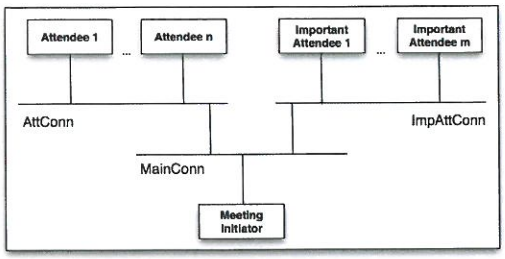

Figura 1. Ejemplo MeetingScheduler.

En la sección 5 se ampliará la descripción de los mecanismos de comunicación indicados anteriormente.

4.2. Esquema básico con JMS

JMS es el servicio de mensajes de J2EE y proporciona la distribución de mensajes siguiendo un esquema productor/consumidor $(1, n)$. El programa que está interesado en recibir mensajes de un tipo (o sea, con un asunto o *topic* en concreto), debe abrir una conexión con el servidor de J2EE. Este servidor será el encargado de enviarle los mensajes solicitados. Los mensajes se entregan de forma asíncrona. En la clase que recibe mensajes se define un método que manejará los mensajes recibidos.

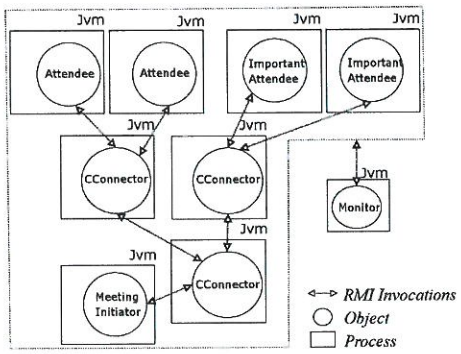

Figura 2. Ejemplo MeetingScheduler con RMI.

Aunque JMS ofrece también comunicación punto a punto, el tipo de comunicación de más interés para nuestra infraestructura es la comunicación *multicast*.

Un esquema para nuestra infraestructura basado en JMS podría ser el que sigue: los conectores de C2 se traducen en canales o *topics* de JMS, de tal forma que si dos componentes estuvieran conectados por un conector en la arquitectura C2, la implementación serían dos procesos Java ejecutando cada uno en su propia máquina virtual y subscritos a un mismo *topic*, que usarían para comunicarse enviando *requests* y *notifications*. El tipo de mensaje en el que estaríamos interesados de entre los que ofrece JMS es el que transporta un objeto, que se denomina *ObjectMessage*. Con ese tipo de mensaje podríamos encapsular en un objeto operaciones y parámetros.

Una ventaja de JMS es que los objetos Java que se comunican entre sí no están obligados a mantener referencias, como en RMI. Esta ventaja permite un mayor desacoplamiento entre los componentes, siendo que esta propiedad es fundamental para el desarrollo de sistemas distribuidos. Los mensajes de JMS se distribuyen como eventos entre los procesos subscritos a cada tipo de mensaje. Por tanto, se mantendría el anonimato entre los componentes. En ese sentido, no necesitaríamos procesos que actuasen como conectores C2, como es el caso de la implementación con Java RMI.

Sin embargo, aparecen dos problemas. El primero se da en las arquitecturas en las que existen conectores conectados entre sí: necesitamos usar un objeto adicional para enviar las peticiones entre los conectores, reenviando los mensajes de JMS a los *topics* correspondientes. El segundo problema es que un conector C2 puede aplicar distintas políticas de filtrado a las *requests* y a las *notifications*. Una política de filtrado *no filtering*, mediante la cual los mensajes le pueden llegar a un componente aunque este no los entienda, encaja perfectamente con la implementación que hemos descrito. Sin embargo este esquema simple no nos serviría para obtener una política *message filtering*, mediante la cual un mensaje nunca le debería llegar a un componente que no la entienda. Por esta razón habría que buscar otras soluciones, como implementar el filtrado de los conectores mediante otro objeto, encargado de gestionar y distribuir los mensajes que se envían a través de los distintos *topics* que implementan los conectores. También será necesario un *topic* común para todos los componentes que sirva para enviar mensajes de control y reconfiguración.

Ejemplo del Meeting Scheduler con JMS Dada una política de filtrado *no filtering*, supongamos que el componente Meeting Initiator envía una *request* solicitando las fechas preferentes de los asistentes. Esa *request* se traducirá en un mensaje enviado al topic llamado MainConn (ver figura 3). En dicha figura se indican los *topics* a los que están subscritos cada uno de los elementos arquitectónicos. Como se ha comentado anteriormente, el proceso monitor, que posee una descripción de toda la arquitectura, es el encargado de enviar peticiones entre conectores que están directamente conectados. Por esta razón y dado que el conector MainConn está conectado a los conectores ImpAttConn y AttConn, el monitor reenviará el mensaje por sus topics (nombrados de la misma manera). De esta forma, los componentes ImportantAttendee y Atteende, que están conectados a los topics ImpAttConn y AttConn respectivamente, recibirán el

mensaje y lo procesarán, enviando posteriormente una *notification* con la información solicitada. El ejemplo descrito en la figura 1 de este documento podría ser implementado con Java JMS siguiendo el esquema expuesto en la figura 3. Los mensajes para la configuración del sistema se envían a un topic llamado Control, al que todos los componentes del sistema estarán subscritos.

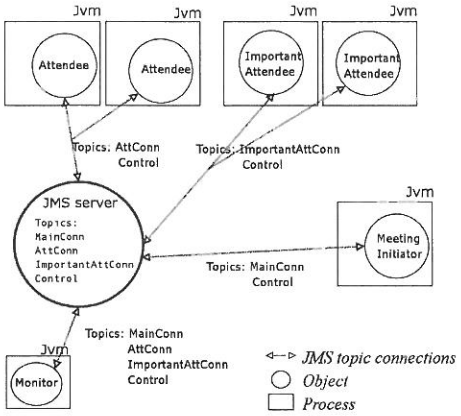

Figura 3. Ejemplo MeetingScheduler con JMS.

5. Infraestructura para la comunicación entre componentes C2 utilizando Java RMI

En esta sección se presenta en detalle la infraestructura que hemos desarrollado para soportar la comunicación entre componentes C2 utilizando Java RMI. Como hemos dicho en la sección anterior, los componentes procesan las peticiones almacenándolas en una cola y devolviendo el control inmediatamente, de tal forma que la invocación por RMI se comporte como una llamada asíncrona. Para la obtención de información acerca de los componentes se usarán los mecanismos de introspección de Java Beans. Después de usar la introspección, los componentes anuncian los servicios que ofrecen a los conectores.

La política de filtrado que utiliza un conector cuando se comunica con un componente C2 es la denominada *message filtering*. Hemos elegido esta política de filtrado debido a que la entrega de las peticiones es eficiente, ya que esta política obliga a que los mensajes sólo lleguen a los componentes que los entienden. Por tanto, una petición de servicio sólo llegará a un componente que implementa dicho servicio. Sin embargo, para la comunicación entre conectores se ha implementado una política *no filtering*. Esto significa que habrá peticiones que lleguen

a conectores que no tienen conectado ningún componente o conector que pueda procesarlas. Cambiar la política de filtrado en los conectores supondría un aumento en la complejidad, ya que habría que propagar toda la información sobre los servicios ofrecidos y los servicios requeridos por los componentes a través de toda la arquitectura (es decir, habría que realizar el cierre transitivo sobre los conectores).

Se ha desarrollado un prototipo que se compone de tres clases principales:

- CComponent: implementa los servicios genéricos de un componente de C2. Un componente tiene que ser capaz de emitir *requests* y *notifications* por sus conectores sin saber el componente que los va a procesar. Cualquier componente que utilice esta infraestructura tendrá que heredar de esta clase abstracta CComponent. La clase CComponent implementa una interfaz llamada IComponent, que se usará para hacer la conversión de las referencia a objetos remotos.
- CConnector: implementa los servicios genéricos de un conector C2. Es el encargado de encaminar las *requests* y las *notifications* hacia los componentes que las deben procesar, actuando básicamente como un intermediario entre el componente que necesita un servicio y el que lo proporciona. Los conectores deben ser capaces de interrogar a los componentes que tienen conectados y saber los servicios que ofrece cada uno. Cualquier conector que utilice esta infraestructura deberá ser instancia de esta clase CConnector. Esta clase implementa a su vez una interfaz llamada IConnector utilizada para hacer la conversión de referencias remotas.
- Monitor: es la clase de los objetos encargados de configurar la arquitectura y enlazar los distintos componentes usando los conectores. Un monitor está dotado de un interprete de comandos que servirá para modificar la arquitectura de componentes de forma dinámica. Tanto CComponent como CConnector implementan una interfaz llamada IElement, que define las operaciones para arrancar y parar los elementos del sistema, usadas por el objeto monitor.

La relación entre estas se detalla en el diagrama de clases de la figura 4(a). En la figura 4(b) se ilustran las clases que implementan los componentes del ejemplo Meeting Scheduler y los tres conectores necesarios.

5.1. Conectores

Un conector debe ofrecer métodos para conectarse a otros elementos (componentes y/o conectores). Una vez conectado a un elemento, el conector tiene que mantener el enlace, que se traduce en una referencia a un objeto remoto. Como parte de las referencias a los componentes conectados a él, mantiene una lista con todos los servicios ofrecidos por los componentes situados por encima para resolver las *requests* y otra lista con los servicios ofrecidos por los componentes situados por debajo para resolver las *notifications*. Estas tablas se llamarán Requests y Notifications respectivamente. Además, un conector mantiene otras dos

listas de referencias, una para los conectores conectados en su dominio *top* (TopConnectors) y otra para su dominio *bottom* (BottomConnectors). El esquema presentado en la figura 5 representa estos enlaces.

El conector posee dos colas en las que se van almacenando los mensajes que contienen peticiones. Los mensajes se almacenan en las colas mediante invocaciones remotas que devuelven el control inmediatamente al objeto invocador. De esta manera, se proporciona un comportamiento asíncrono a las llamadas RMI tal y como se explicó en la sección 4.1. Todo mensaje recibido por la cola de la parte inferior del conector (BottomQueue) será una *request*, por tanto al procesarla se deberá consultar la tabla de *requests* para comprobar si algún servicio corresponde con la petición. Si eso ocurre, el conector será el encargado de invocar al objeto remoto que implementa el servicio para enviarle el mensaje. Una vez hecho esto, se procederá a reenviar el mensaje a todos los conectores de la tabla TopConnectors. Los mensajes recibidos en la cola superior (TopQueue) se procesarán de forma análoga a la descrita para la cola de la parte inferior.

Los dos threads que procesan las colas estarán inactivos mientras que estas estén vacías. En cuanto entre un mensaje en la cola, el thread asociado entrará en ejecución. Los conectores deben ofrecer métodos para conectarse a otros conectores y a componentes. El objeto remoto encargado de invocar esos métodos y enlazar los componentes del sistema es el objeto monitor.

En el conector existen además dos tablas, RequireFromTop y RequireFromBottom (ver figura 5), en las que se anotan todos los servicios de los que dependen los componentes que están conectados a él, tanto para *requests* como para *notifications*. Estas tablas se rellenan cuando un componente se conecta a un conector, igual que las tablas Requests y Notifications. El conector ofrece un método llamado CheckDependencies para comprobar que todos los servicios requeridos son ofrecidos por los otros componentes a los que este tiene acceso (componentes conectados a conectores por los que se propagan las peticiones).

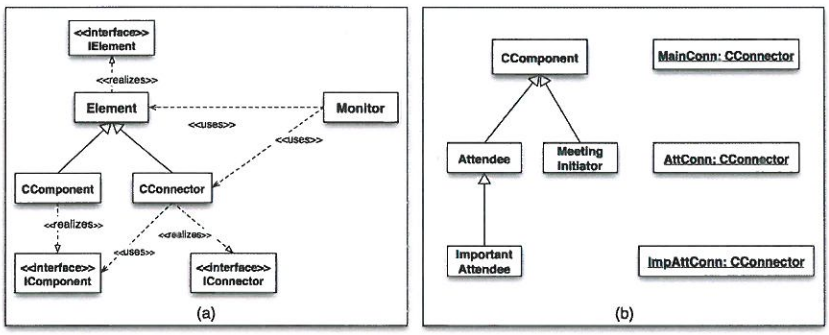

Figura 4. (a) Diagrama de clases de la infraestructura con RMI. (b) Diagrama del ejemplo Meeting Scheduler.

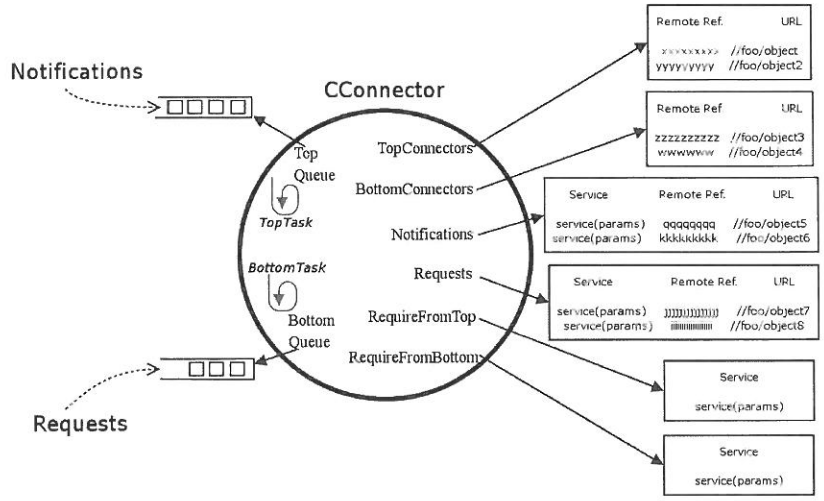

Figura 5. Esquema de un conector.

Este método será invocado por el monitor cuando arranque el sistema para comprobar que todas las dependencias se resuelven.

El método CheckDependencies sigue el siguiente algoritmo. Para comprobar las dependencias de servicios requeridos por el *top* (por tanto de *requests* requeridas por componentes conectados por debajo al conector), lo primero que hace es comprobar si esos servicios están en su propia tabla Requests. Si un servicio está en su propia tabla de Requests, lo marcará como provisto directamente. Si no, enviará una *request* especial llamada CHECK a sus conectores de la parte superior preguntando por componentes que proporcionen el servicio. Los conectores por los que se propaga el mensaje revisarán sus tablas Requests, y si encuentran un servicio que coincida con el requerido, enviarán un mensaje de respuesta especial llamado RCHECK en forma de petición de servicio hacia abajo (*notification*). Todos los conectores por los que se propaga el RCHECK revisarán sus tablas RequireFromTop, y si encuentran un servicio que coincida con la descripción del mensaje, lo marcarán como provisto. La forma de comprobar las dependencias de los servicios requeridos por los componentes conectados por la parte superior de un conector se hará de forma análoga a la descrita.

Una vez que se ha aplicado el algoritmo en los conectores, se revisarán las tablas RequireFromTop y RequireFromBottom en busca de algún servicio marcado como no provisto. Si se encuentra alguno, el componente no habrá pasado la prueba y devolverá al monitor una lista con los servicios requeridos que no se le ofrecen.

En el diseño de los componentes y en el de los conectores existen threads concurrentes. Para asegurar su correcta ejecución, se han usado los mecanismos

de sincronización de Java para garantizar la exclusión mutua en las regiones críticas. La exclusión se implementa mediante regiones críticas sobre los objetos que contienen los datos compartidos. Nótese que la exclusión implementada con regiones críticas proporciona una mayor eficiencia que la sincronización de los métodos de la clase.

5.2. Componentes

Los componentes del sistema heredan de una clase abstracta llamada CComponent que proporciona los mecanismos para interactuar con los demás elementos del sistema. En la figura 6 se muestra el esquema que siguen los componentes.

La acción de conectar un componente a un conector no es responsabilidad del componente. El encargado de enlazar conectores con conectores y componentes con conectores es el monitor. Pero, al igual que los conectores, los componentes tienen que ofrecer mecanismos para dejarse conectar con esos elementos. En el caso de los componentes, estos mecanismos son dos métodos que fijan la referencia al conector de la parte superior (TopConnector) y el de la parte inferior (BottomConnector).

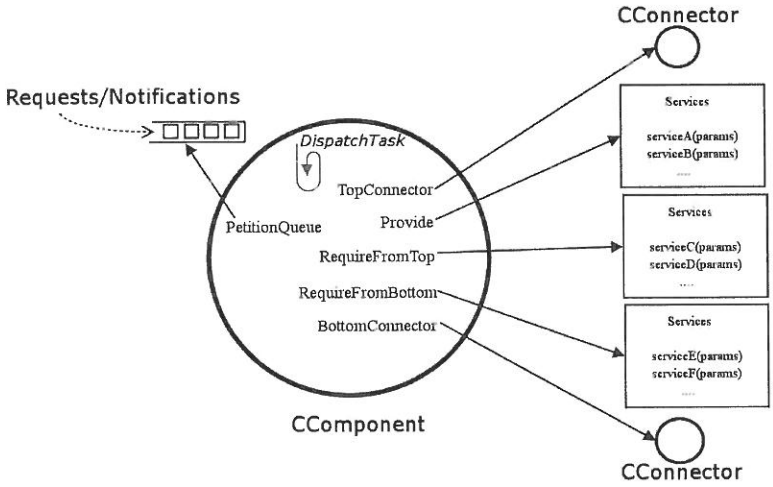

Figura 6. Esquema de un componente.

El componente tiene una cola donde los conectores depositan las peticiones mediante una invocación RMI. El componente tiene un thread de ejecución pendiente de recoger las peticiones de la cola y procesarlas. Este thread se mantiene dormido mientras la cola de peticiones esté vacía. A parte de ese thread, el componente podrá tener tantos threads de ejecución como sean necesarios para el

servicio que debe ofrecer. Esta clase mantiene una tabla llamada Provide con todos los métodos que pueden ser invocados en una petición, esto es, con los métodos que implementan los servicios que ofrece el componente. Esta tabla se rellena usando la API de introspección de Java Beans cuando el componente ejecuta su método constructor. En esta tabla se insertan todos los métodos públicos que posea la clase que hereda de la clase abstracta CComponent. Si tenemos componentes que heredan a su vez de otros componentes, se incluirán todos los métodos públicos sin valor de retorno de la jerarquía de clases hasta llegar a la clase CComponent. La tabla de métodos accesibles contendrá descriptores de métodos de Java. Al conector se le pasará una copia de la tabla con los nombres y los parámetros en formato de cadena de caracteres y sin los descriptores de los métodos. Los parámetros de las peticiones tienen que concordar con los descriptores de método para que puedan ser ejecutados.

Un componente también mantiene dos tablas llamadas RequireFromTop y RequireFromBottom, con los servicios (*requests* y *notifications* respectivamente) que necesita que le ofrezcan los otros componentes para funcionar correctamente. Las tablas RequireFromTop y RequireFromBottom se rellenan en el método constructor del componente, adquiriendo la información de un fichero de configuración. En la implementación actual, toda clase que implemente un componente deberá tener un fichero de configuración con la información de los servicios requeridos por un componente, tanto por su *top* como por su *bottom* . Sin un fichero de configuración correcto, un componente no podrá ser arrancado. Una vez que ha heredado de la clase abstracta CComponent, la clase que implementa el componente puede invocar métodos para enviar *requests* y *notifications* a los conectores a los que está conectado:

```
protected void sendRequest(String name, Vector params)
protected void sendNotification(String name, Vector params)
```

Esos métodos se encargan de encapsular los datos en un solo vector y de encaminar la petición al conector adecuado. De esta forma se consigue que las peticiones viajen hacia arriba o hacia abajo en la arquitectura de componentes C2 llegando a los componentes que implementan el servicio requerido.

5.3. Monitor

El monitor es el encargado de configurar el sistema. Este objeto ordenará conexiones y desconexiones a los elementos del sistema mediante la invocación remota de métodos. También ofrecerá al usuario la topología de la arquitectura y la posibilidad de parar y arrancar los elementos del sistema.

El sistema puede tener sus elementos (componentes y conectores) distribuidos en diferentes máquinas físicas. Cada uno de esos elementos estará registrado en el registro RMI de su máquina local. El objeto monitor es el encargado de consultar todos los registros de las máquinas involucradas y mantener un listado con las URLs de todos los elementos del sistema. Cuando el monitor ordena a un conector que se conecte a un componente, le pasa como parámetro la URL

de este. El objeto monitor adquiere todos los registros RMI que podrán contener componentes o conectores de un fichero de configuración llamado Monitor.conf. Mediante el uso de un patrón de nombrado en los distintos elementos del sistema, el monitor puede localizar a los componentes y conectores dados de alta en los distintos registros de RMI. Además, el monitor no mantiene estado (*stateless*), lo que entre otras cosas significa que no mantiene la actual topología arquitectónica; cuando precisa información sobre cualquier elemento arquitectónico o la propia topología la obtiene interrogando a los propios elementos. Esta propiedad, aunque disminuye la eficiencia del monitor, permite tolerancia a fallos: si el monitor cae puede crearse otro.

El monitor es el único elemento del sistema que se comunica con todos los demás elementos, ya sean componentes o conectores. Este elemento no mantiene referencias remotas a los elementos del sistema, sino que cada vez que necesita establecer comunicación con un elemento resolverá el nombre en el registro de RMI necesario.

En la implementación actual, el monitor ofrece un intérprete de comandos que permite la configuración del sistema. Estos comandos se obtienen de la entrada estándar del proceso. Por esta razón, los comandos se pueden obtener también a partir de otros procesos o ficheros mediante el uso de *pipes* o redirecciones.

6. Conclusiones y trabajos futuros

En este trabajo se ha presentado un conjunto de clases de Java que ofrecen la posibilidad de crear componentes independientes que se ejecutan como procesos pesados y que interactúan entre sí mediante RMI. Estas clases nos dan la posibilidad de crear un sistema de componentes siguiendo el estilo arquitectónico C2. Tanto para la construcción como para la ejecución de los componentes tan sólo es necesaria la tecnología ofrecida en la plataforma J2SE. Los componentes de la arquitectura se pueden ejecutar en distintas máquinas físicas, ya que toda la comunicación está basada en Java RMI. Las clases que se ofrecen son transparentes para los programadores de componentes: los programadores tan sólo deben programar clases de Java que hereden de una clase que implementa toda la maquinaria necesaria para interactuar con el sistema. Además deben crear un pequeño fichero de configuración especificando los servicios de los que depende su componente, con el fin de detectar incoherencias en el sistema de componentes.

El sistema desarrollado se configura mediante el uso de un monitor que ofrece un intérprete sencillo de comandos. El monitor además permite la reconfiguración dinámica de la arquitectura. Este monitor se puede mejorar incorporándole una interfaz gráfica, mediante la cual se pueda configurar el sistema de una forma más cómoda conectando los elementos gráficamente.

Como posible mejora se podría incorporar al sistema un compilador del lenguaje de descripción de arquitecturas C2SADEL [6]. Ese elemento transformaría descripciones de arquitecturas escritas en dicho lenguaje a esqueletos de código Java con todo lo necesario para que el programador tan sólo se preocupara de

añadir la lógica a los componentes, y no tuviera que describir la arquitectura mediante el monitor.

Por otro lado, y a corto plazo, vamos a diseñar e implementar la política de filtrado *message filtering* con Java JMS para resolver problemas indicados en la sección 4.2.

Finalmente, queremos hacer notar que uno de los objetivos de la infraestructura es dar soporte para el desarrollo de un sistema de componentes que proporcionen los bloques básicos de construcción para sistemas tolerantes a fallos con distintas calidades de servicio. Una vez acabado el diseño y la implementación de la infraestructura, se comenzará a desarrollar los componentes que ofrecen servicios de tolerancia a fallos, como componentes de consenso, de detección de fallos o de gestión de grupos.

Referencias

1. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice.* Addison-Wesley, Reading, MA., 1998.
2. B. Eckel. *Thinking in Java, 3rd ed. Revision 4.0*, chapter 13: Concurrency. November 2003.
3. I. Muñoz Fernández, J. E. Pérez Martínez, and E. Soriano Salvador. Comunicación intercomponente en ArchStudio 3.0: diseño del conector "message filtering". *XII Jornadas de Concurrencia y Sistemas Distribuidos*, 2004.
4. Institute for Software Research of the Californa University (Irvine). Archstudio: A software architecture-based development environment.
5. IEEE. *Std. 1471-2000 Recommended Practice for Architechtural Descriptions of software-intensive Systems.* IEEE, 2000.
6. N. Medvidovic. *Architecture-Based Specification-Time Software Evolution.* PhD thesis, University of California, Irvine, 1999.
7. N. Medvidovic and D.S. Rosenblum. Assesing the suitability of a standard design method for modeling software architectures. *Proceedings of the First Working IFIP 52 Conference on Software Architecture (WICSA1)*, February 1999.
8. Sun Microsystems. *JavaBeans^{tm}*. Palo Alto, CA, 1997.
9. Sun Microsystems. *Java^{tm} Message Service (Version 1.0.2).* Palo Alto, CA, 1999.
10. Sun Microsystems. *Java^{tm} Remote Method Invocation Specification (Version 1.4).* Palo Alto, CA, 2002.
11. Sun Microsystems. *EnterpriseJavaBeans^{tm} Specification (Version 2.1).* Palo Alto, CA, 2003.
12. M. Shaw and D. Garlan. *Software Architecture. Perspectives on an Emerging Discipline.* Prentice-Hall, Upper Saddle River, NJ, 1996.
13. R. Taylor, N. Medvidovic, K. M. Anderson, J. Whitehead, J. E. Robbins, K. A. Nies, P. Oreizy, and D.L. Dubrow. A component- and message-based architectural style for GUI software. *IEEE Transactions on Software Engineering, vol. 22, no. 6, pp. 390.*, June 1996.
14. A. van Lamsweerde, R. Darimont, and P. Massonet. *The Meeting Scheduler System: Preliminary Definition.* Unité d'informatique, B-1348 Louvain-la-Neuve (Belgium), 1999.